Modeling wire delay, area, power, and performance in a simulation infrastructure

N. P. Carter
A. Hussain

We present Justice, a set of extensions to the Liberty simulation infrastructure that model chip area, wire length, and power consumption. Based on an architectural specification of a processor, Justice estimates the area and per-access power consumption of each module in the architecture. It then constructs a floorplan for the processor and computes the length and delay of critical communication paths. Finally, Justice modifies the architectural specification to account for wire delay and generates an executable simulator for the processor. To illustrate its capabilities, we simulate a number of very long instruction word (VLIW) architectures. Our results illustrate how Justice makes it possible for designers to compare the costs and benefits of different changes to an architecture and demonstrate the importance of considering wire delay early in the design process.

Introduction

As fabrication technology advances, it is rapidly becoming impossible to evaluate the design of a microprocessor or other integrated circuit without considering how that circuit will be implemented. Wire delay has become a major component of overall performance [1, 2], making it necessary for designers to understand how the modules in their design communicate and how the placement of modules affects the communication delay between other modules. Similarly, power consumption is becoming a key factor in both portable and desktop systems, affecting both battery life and the cost and feasibility of cooling a processor.

Given the impact that these implementation-level effects have on the suitability of an architecture for a given implementation, it is critical that designers have feedback about them early in the design process, when major changes to the architecture are still possible. In this paper we present Justice, a set of extensions to the Liberty [3] simulation infrastructure that model the area, power consumption, and critical wire lengths of an architecture. Justice allows designers to consider the tradeoffs among performance, power, and implementation costs in their designs and to compare the effectiveness of widely varying processor architectures.

To simulate a microprocessor in Liberty, a designer creates an architectural description file (ADF) that describes the processor. Liberty takes the ADF as an input and generates an executable simulator for the architecture. During simulator compilation, Justice parses the ADF and estimates the area and power consumption per access of each module in the architecture. It then creates a processor floorplan and computes the physical length and wire delay of each communication channel on the chip.

Justice inserts first-in first-out (FIFO) delay queues into each communication channel whose wire delay is greater than the cycle time of the processor. It then adds activity counters to each module to record how often the module is accessed during program execution. After the user simulates the execution of a program on the simulator generated by Liberty–Justice, Justice executes a postprocessing pass that generates per-module and overall power consumption estimates based on the output of the activity counters.

Our discussion of Justice begins with an overview of the Liberty simulation environment followed by a description of Justice and our power, area, and wire models. To evaluate Justice, we model a variety of very long instruction word (VLIW) processors, illustrating the tradeoffs among area, power consumption, and performance in these architectures. We simulate our architectures at a wide range of clock rates to illustrate the impact that wire delay has on performance as clock rate increases. Finally, we discuss related work and conclude the paper.

0018-8646/06/$5.00 © 2006 IBM

Liberty overview

The Liberty simulation environment (LSE) [3–5] is a toolkit that allows architects to develop high-performance simulators for a wide range of processor architectures. Unlike many simulators such as SimpleScalar [6], Liberty does not directly model a particular style of architecture. Instead, it provides a framework that generates simulators for computer architectures based on configurations provided by the user.

Liberty models an architecture as a set of *modules* that represent functional blocks within an architecture. A module consists of a set of *ports* that define its interface and the code that defines module behavior. Modules communicate through *channels* that represent communication paths in the architecture. Channels model communication at an abstract level, rather than cycle by cycle, using a request and reply protocol to pass data structures between modules. Each data structure represents an event in the simulation and contains enough context information that changing the delay of a channel does not affect the results of a simulation.

The LSE provides a set of predefined modules that model common architectural blocks, such as execution pipelines and memory arrays. Novel features of an architecture can be modeled either as a collection of these predefined modules or by creating custom models that directly simulate the feature in question. Liberty also provides a number of emulator modules that simulate the execution of various common instruction sets. These emulators significantly reduce the effort required to implement a simulation of a processor that executes a conventional instruction set architecture (ISA), because the user can rely on the emulator to correctly execute programs and focus their effort on implementing an accurate performance model of the processor.

Liberty users model architectures by creating ADFs that instantiate the modules and channels that make up the architecture. The Liberty simulator constructor, BuildSim, takes an ADF as its input and generates an executable program binary that simulates the architecture it describes. Since a Liberty-generated simulator is a customized executable (rather than a generic simulator that interprets a configuration file at runtime to extract simulation parameters), Liberty achieves very high simulation speed, increasing the size and number of the applications that can be used to test an architecture.

Once Liberty has generated a simulator for an architecture, users simulate programs by executing the simulator binary with the program to be run on it as an argument. Basic Liberty simulations determine the number of clock cycles an architecture requires in order to simulate a given program. In addition, Liberty users may define custom event-monitoring routines, known as *data collectors*. Data collectors defined by a user become part of the configuration of an architecture and are invoked whenever the event or events specified in the data collector occur. Because data collectors are user-defined, they can be as simple or complex as the user requires.

Liberty was chosen as the basis for Justice because of its flexibility and explicit representation of communication paths. Its module-and-channel representation allows it to model a wide range of processor architectures, from traditional superscalar and VLIW to clustered and gridded. This provides much more flexibility than does a simulator that assumes a particular style of architecture. Similarly, the use of communication channels to pass data among modules allows Justice to identify the important communication paths in an architecture, simplifying the process of floorplanning and wire-length calculation. Finally, the semantics of the channel-based communication model allow Justice to change the timing of communication paths that represent long wires without affecting the correctness of the simulation, which greatly simplifies modeling of wire delays.

Justice extensions to Liberty

Figure 1 shows how Justice works with Liberty and the power models provided by TEM^{2}P^{2}EST [7] to generate power, area, wire-length, and performance estimates for an architecture. Liberty–Justice uses two input files to generate a simulator for an architecture: an ADF for the design, which is unchanged from the base LSE, and a technology description file (TDF) for the fabrication process that the simulation should use.

Justice adds two components to BuildSim: a power constant calculator and a physical approximation generator (PAG). The power constant calculator is relatively independent of the remainder of BuildSim. It parses the ADF and TDF to determine the parameters of each module and then calls TEM^{2}P^{2}EST to generate active and idle power consumption estimates for each module. These estimates are saved with the simulator generated by BuildSim for use by the postprocessing power aggregation step.

TEM^{2}P^{2}EST supports both analytical and empirical power models. Wherever possible, Justice uses the analytical power models, because they are expected to be more accurate than the empirical models across a wide range of fabrication processes. The empirical power models of TEM^{2}P^{2}EST are used only on modules such as arithmetic logic units that are difficult to model analytically. The interface between Justice and TEM^{2}P^{2}EST is designed to be extremely modular, allowing architects to substitute other power models if the TEM^{2}P^{2}EST models are not appropriate for their application.

The Justice PAG is integrated into BuildSim and modifies the architecture defined by the user to account for wire delays and to track the activity of each module in the design. The first build phase of BuildSim generates a netlist of the modules and channels in an architecture, which the PAG parses to determine the connections among modules and to generate an area estimate for each module. Justice models the height, width, and total area of each module using a combination of analytic and empirical techniques. Random access memory (RAM)-based modules—including caches, queues, register files, and translation lookaside buffers (TLBs)—are modeled analytically, taking into account both the base area required for each RAM cell and the area required for read and write ports. When this model does not apply, Justice uses empirical area models that were generated by measuring the area of similar units in existing microprocessors and scaling to match the simulation feature size.

Using the area estimates for each module, Justice generates a floorplan for the architecture, treating each module as a fixed block whose aspect ratio cannot be changed, although blocks can be rotated in order to generate a better floorplan. Justice uses a simulated annealing algorithm to optimize the floorplan for a combination of total area and wire length. In future work, we plan to make this optimization criterion accessible to the user and to allow the user to designate certain communication channels as *performance-critical* in order to encourage floorplans that minimize the length of those channels.

Once the floorplan is complete, Justice estimates the wire length of each communication channel that connects two or more modules in the architecture as the Manhattan distance[1] between the centers of those modules. Justice then estimates the wire delay along each channel using the properties of the fabrication process being modeled and divides this delay by the simulated cycle time to determine how many cycles of delay each communication channel will incur, rounding to the nearest cycle.

Justice next modifies the netlist, adding FIFO queues of the appropriate depth to each communication channel whose wire delay is one or more cycles. These queues model the wire delay along the channel, making the assumption that long wires are pipelined to allow multiple data values to be in transit simultaneously. Justice determines the latency of each communication channel independently, which may lead to different numbers of delay cycles being added to different paths through an architecture. In a hardware implementation of the

[1] The "Manhattan distance" between two points is the sum of the (absolute) differences of their coordinates. Informally, it is like measuring distance in a city (e.g., Manhattan) in city blocks rather than as the crow flies.

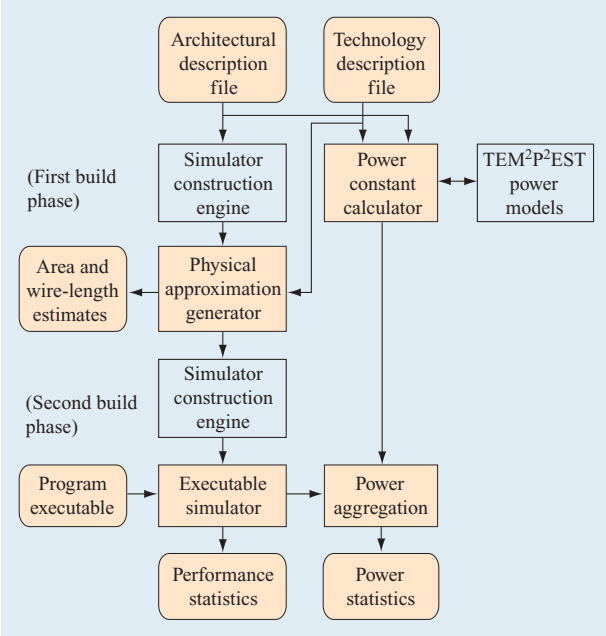

Figure 1

Simulation with Liberty and Justice. (The unshaded boxes are functions performed by tools other than Justice.)

architecture, this would lead to incorrect operation because the inputs to modules would arrive on different cycles, and it would be necessary to add extra pipeline registers to some wires to balance the delays along them.

In a Liberty–Justice simulation, the protocol used by the communication channels prevents uneven delays from affecting the correctness of a simulation. Modules do not generate their outputs until all of their inputs are ready. Values that arrive early are queued by the simulator until they are used, and the progress of the simulation is determined by the longest path through the design, exactly as it would be if the delays along each path were balanced, which greatly simplifies the task of the PAG.

After area and wire delay estimation, the PAG adds data collectors to each module that count the number of times the module is accessed during simulation. It then outputs information about the area of the design and critical wire lengths, and passes the modified netlist to the second build phase of BuildSim, which generates an executable simulator for the architecture.

Users run a program on a Liberty–Justice simulation in the same way they would run a program on a baseline Liberty simulation: by invoking the simulation executable with the program as an argument. The modified simulator executes the program and outputs performance statistics, such as the number of cycles required for the

simulation, the activity counts generated by the data collectors inserted by Justice into each module, and the results of any user-defined data collectors. This output is passed to the Justice power aggregation code, which uses the activity counts from each module to determine the number of active and idle cycles for the module and multiplies these values by the active and idle module power estimates to compute the total energy consumed in the module. The power aggregator then sums the energy consumed in the modules to generate energy and power consumption estimates for the entire architecture.

Integrating Justice into the Liberty simulation flow in this way significantly extends the capability of the LSE without unduly increasing simulation time and without restricting the types of architectures that can be modeled in Liberty. Justice itself takes very little time to execute, and most of its impact on simulation time comes from the increase in the number of clock cycles that must be simulated when wire delays are taken into account. The modular design of Justice also makes it relatively easy to modify its power or area models, for example, by replacing the floorplanner with one based on a different algorithm or by changing some of the power models. This allows designers to use Justice effectively, even if they feel that one or more of the baseline assumptions made by Justice are not appropriate for their design.

Experimental methodology

To illustrate the capabilities of Justice, we modeled six VLIW architectures in Liberty–Justice: three conventional VLIW processors and three clustered VLIW processors. Our baseline VLIW has eight execution units that share a 32-entry register file. Each execution unit can perform both integer and floating-point computations, and two of the execution units can execute memory operations. We refer to this architecture as the *8W2M* model because it is 8-wide and can perform two memory operations per cycle. We also model an 8-wide VLIW that can perform four memory operations per cycle (8W4M) and a 16-wide VLIW that can perform four memory operations per cycle (16W4M). For our power, area, and wire length studies, we assume that all architectures are fabricated in a 90-nm fabrication process.

Our clustered VLIW architectures divide their functional units into two independent clusters, each of which contains half of the functional units and a 16-entry register file. Operations that execute on a given cluster may access only its register file, although data can be transferred between clusters via explicit move operations. We model three clustered VLIW architectures: an 8-wide processor that can execute one memory reference per cycle from each cluster (8W1M1M), an 8-wide processor that can execute two memory references per cycle from each cluster (8W2M2M), and a 16-wide processor that can execute two memory references per cycle in each cluster (16W2M2M). All of our processor models use 128-KB data and instruction caches. We assume that VLIW instructions are stored in a compressed format, such as the one described in [8], that reduces the amount of space required by no-operation (NOP) operations.

We evaluate these processor architectures using a set of nine benchmark programs taken from the MediaBench suite [9]. Where possible, we used both the encoder and decoder portions of each benchmark, although `rasta`, a speech recognition application, does not have separate encoder and decoder applications. The benchmark programs were compiled for each architecture using the Impact [10] compiler to generate high-quality optimized code.

Results

As a baseline, **Figure 2(a)** shows the average number of instructions per clock cycle (IPC) executed by our architectures on each benchmark when wire delay and clock rate are not taken into account. Overall, IPC numbers are fairly high, as would be expected of highly optimized media applications, with significant variance among applications. Increasing the number of memory instructions allowed per cycle without changing the issue width of the architecture improves IPC by about 4% for both the conventional and clustered VLIW architectures, while increasing issue width from 8 to 16 improves IPC by 8.7% for the conventional VLIW processor, and by 10.5% for the clustered processor.

When wire delay and clock rate are not considered, the IPC numbers of clustered and nonclustered processors with the same number of execution units and memory ports are very similar. On some applications, the nonclustered version of an architecture does slightly better than the clustered version because of intercluster communication overheads in the clustered version. On others, the clustered version achieves higher IPC than the nonclustered version because long-latency operations in one cluster do not stall computation on the other.

Power consumption

Figure 2(b) shows the Justice estimate of the average power dissipated by our architectures when executing each benchmark. These results assume a 90-nm fabrication process and 1.65-V power supply, with all architectures running at 250 MHz, well below the frequency at which wire delays begin to have a significant effect on performance. On average, the 8-wide architectures with four memory ports consume 15% more power than their counterparts with two memory ports, while increasing issue width from 8 to 16 IPC increases power consumption by 60%. Clustered architectures consume slightly less power than their nonclustered counterparts on almost all applications.

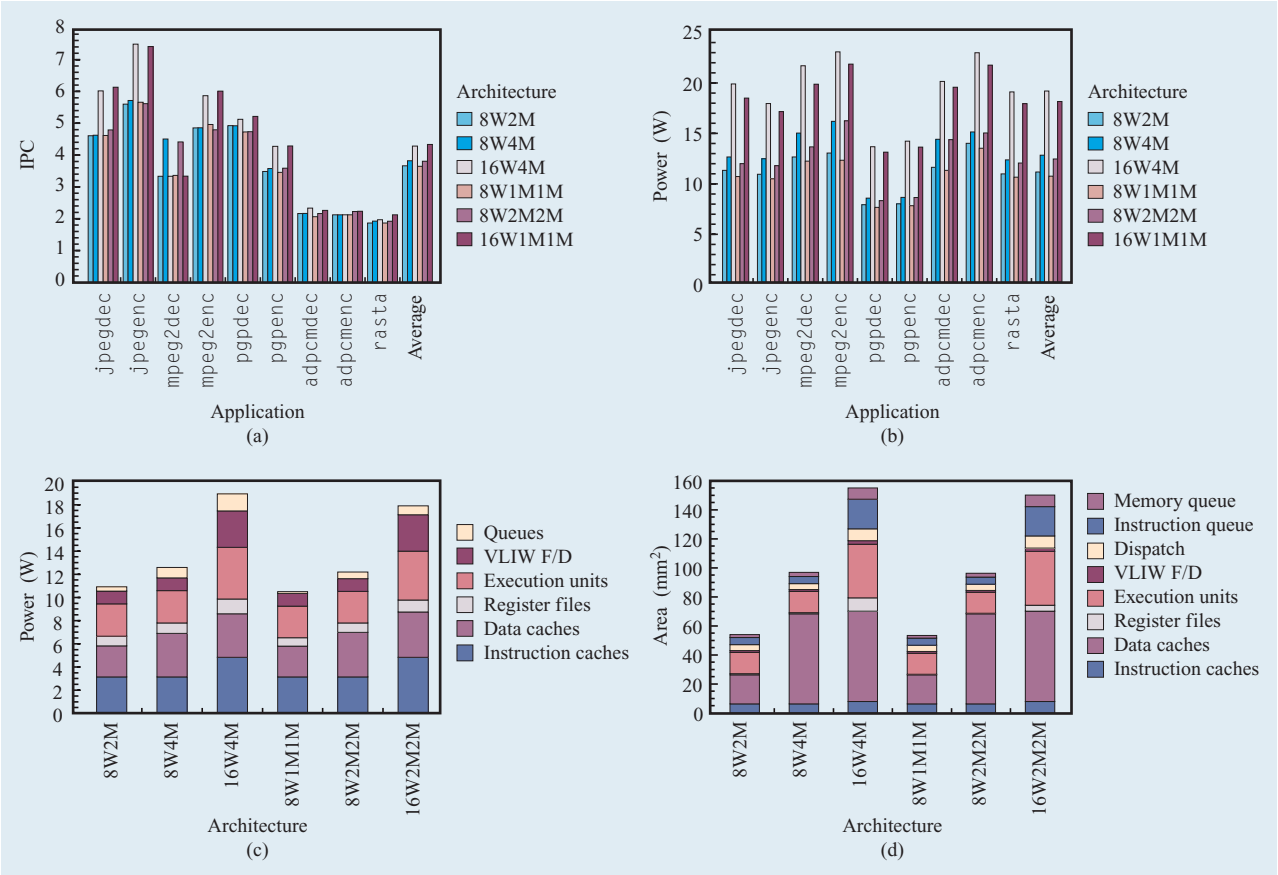

Figure 2

(a) IPC without considering wire length. (b) Power consumption. (c) Breakdown of power consumption by processor module. (d) Area of each modeled architecture.

To give more insight into these results, **Figure 2(c)** shows the contributions of the major sources of power consumption in each architecture to overall power consumption, averaged across all of the benchmarks. The *Queues* category includes the power consumed in the instruction queues, memory queues, and dispatch logic. Other categories include the VLIW fetch and decode (F/D) logic, the execution units, register files, instruction caches, and data caches.

Increasing the number of memory operations that an architecture can execute per cycle from two to four increases data cache power consumption by 43%; it also increases the power consumed in the queues but has relatively little effect on power consumption in other units. Because the data cache consumes about 25% of the power in the base 8W2M processor, these changes explain the overall power differences between architectures with different numbers of memory ports.

The power consumed by the instruction cache is the same for all of the 8-wide architectures and increases by approximately 50% in the 16-wide architectures because of the doubling of the instruction word width. Similarly, power consumption in the execution units varies by only 1% across the different 8-wide architectures but increases by 55–60% when the issue width is doubled. This sublinear increase in execution unit power as instruction width increases is due to the fact that execution units consume significantly less power when idle than when executing valid instructions. Because increasing the issue width yields less-than-linear increases in IPC, it is not surprising that it also causes sublinear increases in power consumption.

Area and wire lengths

Figure 2(d) shows the Justice estimates of the total chip area required to implement each of our architectures in a 90-nm fabrication process and how each of the major

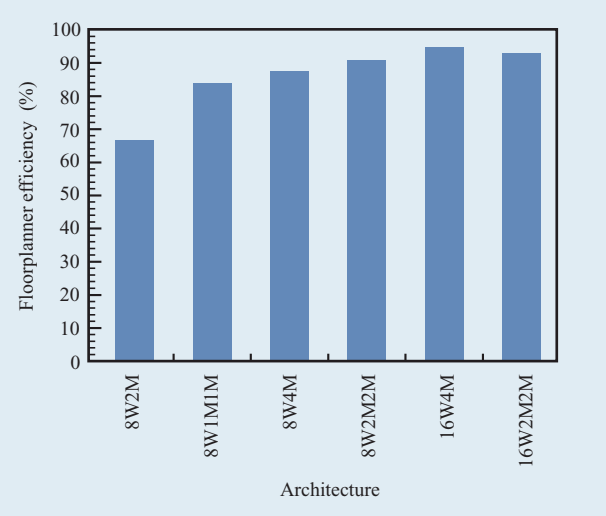

Figure 3

Efficiency of the floorplanner.

architectural structures contributes to the total. The *Queues* category from Figure 2(c) has been subdivided into the area required for the memory queues, instruction queues, and dispatch logic to better illustrate how changes to the architecture affect the size of each of these structures. This graph shows the area of each architecture as the sum of the areas of each of its components, without any blank space that may be introduced during floorplanning.

As would be expected, the area taken up by the execution units grows linearly with the number of execution units in the architecture, while the instruction caches of the 16-wide architectures are approximately 20% larger than those of the 8-wide architectures because of the increase in fetch width in the wider architectures. Similarly, the instruction queues and dispatch logic of the 16-wide architectures are significantly larger than those of the 8-wide architectures. The fetch and decode logic takes up a relatively small fraction of the total area of each of the architectures, illustrating one of the advantages of making the compiler responsible for instruction scheduling.

Because the area of an SRAM bit cell increases quadratically with the number of ports, the data caches of the architectures that support four memory accesses per cycle are approximately three times the size of the data caches in architectures that support only two accesses per cycle. Register file size remains constant across the 8-wide nonclustered architectures but increases when the width of the processor is increased to 16 because of the need to support more ports on the register file. The register files of the clustered architectures are noticeably smaller than those of the nonclustered architectures, since two 16-entry register files are smaller than one 32-entry register file with twice as many ports.

Table 1 Longest wire lengths and delays.

Model	*Wire length* (mm)	*Wire delay* (ns)
8W2M	4.379	0.645
8W4M	4.749	0.759
16W4M	6.685	1.505
8W1M1M	4.169	0.585
8W2M2M	3.928	0.521
16W2M2M	5.276	0.937

Figure 3 shows the efficiency of the floorplanner in placing each of our architectures, measured as the fraction of the area of the placed chip that is taken up by active circuitry. Most of the architectures achieve between 80% and 95% efficiency, with architectures that contain more units generally having higher efficiency. The 8W2M architecture achieves only 66% placement efficiency because of an aspect ratio mismatch between its cache and the remainder of its units. This illustrates one weakness of our floorplanner that we intend to rectify in future work—it treats all modules in the architecture as fixed blocks and is unable to alter the aspect ratios of units, such as cache memories, that are more flexible in their design.

On the basis of its floorplans, Justice computed the length of each wiring channel in our architectures, generating the data shown in **Table 1** for the longest wire length and wire delay in each architecture. As would be expected, the 16-wide architectures have significantly longer wires than their 8-wide counterparts, and the clustered architectures have noticeably shorter wires than their nonclustered counterparts. The one surprise in this data is that the longest wire in the 8W4M architecture is only about 10% longer than the longest wire in the 8W2M architecture, in spite of the fact that Figure 2(d) shows the 8W4M architecture requiring significantly more chip area than the 8W2M architecture. This occurs because the floorplanner does a poor job placing the modules in the 8W2M architecture, leading to a significant amount of blank space in the floorplan. This blank space increases the distance between the centers of the modules in the design, and thus the length of the longest wires.

Impact of wire delay on performance

To illustrate the ability of Justice to model the effect of wire delay on performance, we simulated each of our

target architectures at clock rates ranging from 250 MHz to 5 GHz. All architectural parameters other than wire delay—including fabrication process, pipeline depth, and memory latencies—were held constant across the clock frequencies. **Figure 4(a)** shows the IPC achieved by each of our architectures as a function of clock frequency, averaged across all of our benchmarks. The leftmost *Ideal* column of the graph shows what IPC each architecture achieves when all wire delays are zero cycles, as would occur at extremely low clock frequencies, while the other columns show IPC at their specified clock rates.

This graph illustrates the importance of considering wire delay in estimating the performance of processors. While the 16-wide architectures achieve the highest IPC when wire delay is neglected, they quickly become some of the worst performers when wire delay is considered and the clock rate increased. In general, the IPC of an architecture decreases monotonically as clock rate increases, although some artifacts are introduced because Justice rounds the delay on each wire to the nearest full cycle.

To illustrate the effect of wire delay on absolute performance, **Figure 4(b)** shows the performance of each of our architectures (measured as the product of IPC and clock rate) relative to the performance of the 8W2M architecture at 250 MHz, a clock rate at which none of the architectures see any wire delay effects in our model. In this graph, we can see that the clustered architectures achieve significantly better performance than their unclustered counterparts as clock rates increase, with the 8W1M1M architecture achieving the highest performance at extremely high clock rates.

These results show how Justice can help designers understand the impact that implementation effects will have on their architectures early in the design process. Simulation tools that ignore wire delays can lead designers to choose extremely complex architectures that are difficult to implement at high clock rates, while failing to estimate power consumption can lead to designs that require expensive heat sinks, have poor battery life, or both. By exposing these factors to designers at the start of the design process, Justice allows them to explore a wide range of architectures to select the one best suited to their application.

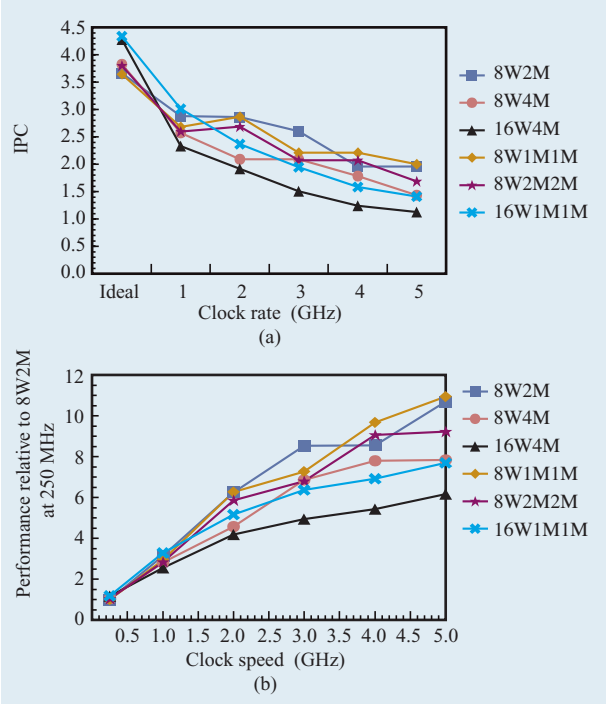

Figure 4

(a) Impact of wire delay on IPC. (b) Impact of wire delay on performance.

Related work

Justice builds on the work of a number of efforts that have studied power modeling and wire effects in architectural simulators. SimplePower [11], Wattch [12], and TEM^{2}P^{2}EST [7] are three examples of tools that have added power modeling to architectural simulations. Each of these tools uses SimpleScalar [6] as its architectural performance simulator and augments SimpleScalar with extensions that model the power consumption of different units within an architecture.

While these three tools are all based on SimpleScalar, they use very different approaches to estimate the power consumed by an architecture. SimplePower relies on an empirically derived table of effective capacitances for each unit in the processor to calculate the total capacitance being charged and discharged on each cycle, and thus the power consumed during the cycle. This approach allows extremely accurate modeling of the power consumed by a particular architecture, particularly one for which accurate layout-based capacitances are available, but the effort required to generate the capacitance table for each new architecture limits the portability of SimplePower.

Wattch, on the other hand, uses parameterizable analytic models to estimate the power consumed by each unit in an architecture, which makes it extremely flexible. TEM^{2}P^{2}EST takes an intermediate approach to power modeling, using a combination of analytic power models and empirical power density estimates to compute the power consumed by each module in an architecture. TEM^{2}P^{2}EST models a processor as a collection of functional blocks and estimates the power consumed in each block on the basis of activity counts generated

during simulation. Both the Wattch and TEM^{2}P^{2}EST power models are modular and flexible enough that they could be applied to Justice. We selected TEM^{2}P^{2}EST on the basis of our ongoing collaboration with the TEM^{2}P^{2}EST authors, which provided significant insight into the tool and how it could be adapted to meet our needs.

Conclusion

Justice is a set of extensions to the Liberty simulation environment that allows designers to estimate, for a given architecture, the power consumption, area, and impact that wire delays will have on performance. During the compilation of a simulator for an architecture, Justice estimates the area of each module in a design and creates a floorplan for the architecture. Using this floorplan, it estimates the physical length of the wires represented by communication channels in the architecture model and adds FIFO delay queues to each channel to simulate its wire delay. Justice also adds activity counters to each module in the architecture and precomputes the amount of power consumed each time the module is accessed. After simulation of a program completes, Justice processes the output of these activity counters to estimate per-module and overall power consumption.

Simulations using Justice demonstrate the importance of considering power and wire delay in architectural simulation. When wire delay was neglected, doubling the number of execution units in an architecture increased performance by approximately twice as much as doubling the number of memory accesses the processor could perform per cycle. However, doubling the number of execution units increased overall power consumption by 60%, while doubling the number of memory ports on the processor increased power consumption by only 15%.

When wire delay and clock rate are taken into account, the benefits of clustering a VLIW architecture become clear, as our clustered architectures significantly outperform their nonclustered counterparts. In addition, the benefits of widening the architecture are called into question. At high clock rates, our 16-wide architectures have lower overall performance than our 8-wide architectures because of the increase in wire length between the execution units and centralized control and memory logic.

While these simulations are somewhat idealized, they clearly illustrate the importance of modeling technology effects when evaluating potential changes to a computer architecture. As wire delay and power consumption become more and more significant with advances in fabrication technology, we believe that simulation tools such as Justice—which enable designers to easily model technology effects in their simulations—will become indispensable tools for computer architects.

Acknowledgments

This work was supported by the Semiconductor Research Corporation (SRC) under Contract No. 785. The material presented here represents the conclusions of the authors and does not necessarily represent the opinions of the SRC. The authors thank David August and the members of the Liberty Research Group at Princeton University for their help in developing Justice, as well as the reviewers and others who have commented on earlier versions of this paper.

References

1. V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures," *Proceedings of the 27th Annual International Symposium on Computer Architecture,* 2000, pp. 248–259.
2. R. Ho, K. W. Mai, and M. A. Horowitz, "The Future of Wires," *Proc. IEEE* **89**, No. 4, 490–504 (April 2001).
3. M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural Exploration with Liberty," *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture,* 2002, pp. 271–282.
4. D. I. August, M. Vachharajani, J. A. Blome, N. Vachharajani, D. A. Penry, and R. Rangan, "Architectural Exploration with Liberty," Tutorial presentation at the 36th Annual International Symposium on Microarchitecture, 2001; see *http://liberty.princeton.edu/News/Tutorials/Micro36/*.
5. The Liberty Research Group; see *http://liberty.cs.princeton.edu/*.
6. D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Technical Report 1342*, Computer Science Department, University of Wisconsin, Madison, WI 53706, 1997.
7. A. Dhodapkar, C. H. Lim, G. Cai, and W. R. Daasch, "TEM^{2}P^{2}EST: A Thermal Enabled Multi-Model Power/Performance ESTimator," *Proceedings of the International Workshop on Power-Aware Computer Systems,* 2000, pp. 112–125.
8. T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings," *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture,* 1996, pp. 201–211.
9. C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture,* 1997, pp. 330–335.
10. W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, "Compiler Technology for Future Microprocessors," *Proc. IEEE* **83**, No. 12, 1625–1640 (December 1995).
11. W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool," *Proceedings of the Asia and South Pacific Design and Automation Conference,* 2000, pp. 340–345.
12. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proceedings of the 27th Annual International Symposium on Computer Architecture,* 2000, pp. 83–94.

Received June 28, 2005; accepted for publication July 26, 2005; Internet publication February 23, 2006

Nicholas P. Carter *University of Illinois at Urbana–Champaign, 1308 W. Main Street, Urbana, Illinois 61801 (npcarter@uiuc.edu)*. Professor Carter is an Assistant Professor at the University of Illinois at Urbana–Champaign (UIUC). Prior to joining the UIUC in 1999, he was a graduate student at the Massachusetts Institute of Technology, where he was the memory system architect for William J. Dally's M-Machine project. Professor Carter's research interests focus on computer architectures that incorporate devices other than silicon transistors and structures other than conventional microprocessors.

Azmat Hussain *Digital Enterprise Group, Intel Corporation, 5200 NE Elam Young Parkway, Hillsboro, Oregon 97124 (azmat.hussain@intel.com)*. Mr. Hussain received an M.S. degree in electrical engineering from the University of Illinois at Urbana–Champaign. He is currently working in the architecture design group for Intel Corporation.